REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-01-

0005

1. REPORT DATE *(DD-MM-YYYY)*
27-11-00

2. REPORT TYPE
Final

3. DATES COVERED
01-9-93 to 31-10-97

4. TITLE AND SUBTITLE
Run-Time Assurance for Distributed Computing Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER
F49620-93-1-0409P00001

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHOR(S)
B. M. McMillin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The University of Missouri-Rolla
Rolla, MO 65409

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Maj. David Luginbuhl
801 N. Randolph St. Rm 732
Arlington, VA 22203-1977

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
unlimited

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

DTIC QUALITY INSPECTED 3

20010109 097

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This work, as an AASERT Augmentation to F49620-92-J-0546, has developed a powerful concept in evaluating formal specifications concurrently with distributed program execution for the purposes of error detection, fault tolerance, and security. This concept is realized in the CCSP evaluation system for axiomatic proofs, for interval temporal formulae, and for a security calculus. We have validated
This concept through nontrivial examples of distributed programs including a dynamic group membership protocol, a distributed database scheduler, of a responsive system modeling railroad trains on intersecting tracks, and of a secure warehouse management system. Moreover, the spinoff technologies from this work, in of themselves have become useful. CCSP can also be used as a debugging tool for distributed programs. Properties used in CCSP can be visualized using abstract glyphs. Both of these achievements may help to bring more use of formal methods into the mainstream.

15. SUBJECT TERMS
Temporal Logic, Responsive Systems, Visualization

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
unclassified	unclassified	unclassified	UL		Bruce M. McMillin
					19b. TELEPHONE NUMBER *(include area code)* (573)-341-6435

Overview

Motivation

Our original goal of the parent project was to find ways to execute program specifications along with the actual program's execution for purposes of run-time assurance - namely for error detection within the scope of fault tolerance. If the execution of the program does not satisfy the specification at run time, then an error has occurred. Since error detection is conceptually the most difficult problem in fault tolerance, this quantification of error detection has proved quite powerful - a system need not rely on hardware or software confidence to avoid or detect errors; the specification provides the absolute truth of correctness.

Actually doing this is difficult even in the sequential environment as one must ask the question ``What is an appropriate level of specification and how does it correspond with the resulting program code?" In the distributed parallel environment with which we are concerned, the challenge becomes greater due to the absence of a globally consistent state in which to evaluate the specification.

The ASSERT funding beyond the parent project's goal was to increase the participation of women in Computer Science while addressing the problems of a run-time system and of visualizing program behavior.

Methodology

The notion of ``the program satisfies the specification' is a powerful abstraction as it immediately draws the researcher into the area of formal logic to express the specification. This, coupled with an existing set of axioms and inference rules for a particular (programming) language provides the appropriate level of representation for run-time error checking. Essentially, the same tools used in program verification are immediately applicable to run-time assurance, namely execution of the proof outline in either a predicate or temporal framework.

Our work provides the run-time semantics to carry out such executions, possibly in the presence of failed hardware and/or software or security intrusions. Nor are we limited to formalized verification systems; our methods work quite well with informally specified assertions. We have developed a set of tools (described below) to carry out these evaluations.

Technical Details

The Axiomatic Approach to Program Verification

The axiomatic approach to program verification is based on making assertions about program variables before, during and after program execution. These assertions characterize properties of program variables and relationships between them at various stages of program execution.

Overall Proof Approach.

Distributed programs are composed of a set of communicating sequential processes. In many programs, it is desirable to save part of the communication sequence between processes. This is done with use of ``dummy" or *auxiliary* variables that relate program variables of one process to program variables of another. In general, to prove properties about the program, first properties of each component process are derived in isolation. These properties are combined to obtain the properties of the whole program using ``global" auxiliary variables; if the proofs do not *interfere*, then this composition is valid. We use Hoare's CSP as a model.

Operational Evaluation of Axiomatic Assertions

Taking an application's proof outline from the verification environment to the distributed operational environment is not a straightforward task. Since assertions may involve global annotations to the program state, we need some way of communicating this state, efficiently. Observing that no state change can influence a CSP process until some communication occurs (since process states are local), we can simply defer update of global state information until an algorithmic communication occurs. Thus, each communication in CSP is augmented with two functions which prepare copies of a processes' global auxiliary variables for communication and unions these variables into a process' local state, respectively. Since we only need to send the most recent copy (and only a newer copy) variables are time stamped with a Lamport clock. Then, the latest copy of each global auxiliary variable is merged with the local processes' state. These communicated auxiliary variables, in turn, along with the sequential processes' state, are what the assertions are evaluated against.

These ideas were embodied in the package CCSP which is reported in:

"CCSP - A Formal System for Distributed Program Debugging," *Programming and Computer Software*, Plenum Publishers, Vol. 21. No. 1, 1995, pp. 45-50, E. Arrowsmith and B. McMillin.

The full CCSP source is available from `http://www. umr.edu/ecl.html`

Program Visualization

The understanding of program behavior is becoming vitally more important now that software is becoming an integral part of industry and everyday life. However, even the best documented code is often not sufficient enough to completely and correctly relay the actual program behavior.
The problem lies beyond being familiar with the programming language and is hidden in the complex mathematics, which govern the program's behavior. This behavior is not easily detected and varies from one program to the next. We propose a method for describing program behavior using two general properties of iterative programs: feasibility and progress. This method can be easily applied to trivial and simple code but an automated tool is required to generate the properties for more realistic code. Therefore, an automated program visualization tool was developed to illustrate the program's behavior in terms of the two properties proposed. *Wheels* take as input program code, reverse engineers the behavior by analyzing the code and then visually relays the extracted information back to the user allowing the user to gain a visual understanding of program behavior. The intent of this research is to use this understanding as a means of learning and teaching as well as a means for providing run time assurance to check the expected behavior.

This work is reported in

"Wheels: An Automated Program Analysis Tool," *The 8th International Conf. on Software Engineering and Knowledge Engineering,* June 10-12, 1996, Lake Tahoe, NV, pp. 269-276 (with A. Sun).

Student Participants

The following students were supported during the lifetime of this grant.

Name	Topic	Status
Elizabeth Arrowsmith	Developed the CCSP System	Left the Program to pursue Ph.D. at Washington University
Aggie Sun	Declarative Approach to Generalizing the Understanding of Program Behavior through Program Visualization	Ph.D. 1996

Contribution

We feel we have developed a powerful concept in evaluating formal specifications concurrently with distributed program execution. Moreover, the spinoff technologies from this work, in of themselves have become useful. CCSP can also be used as a debugging tool for distributed programs. Temporal Subsumption functions as a quick and powerful proof checker for Hoare triples. Both of these achievements may help to bring more use of formal methods into the mainstream.

It is unfortunate that, despite a one-year extension, no suitable women Ph.D. students wer found and the 3rd year funding had to be turned back.